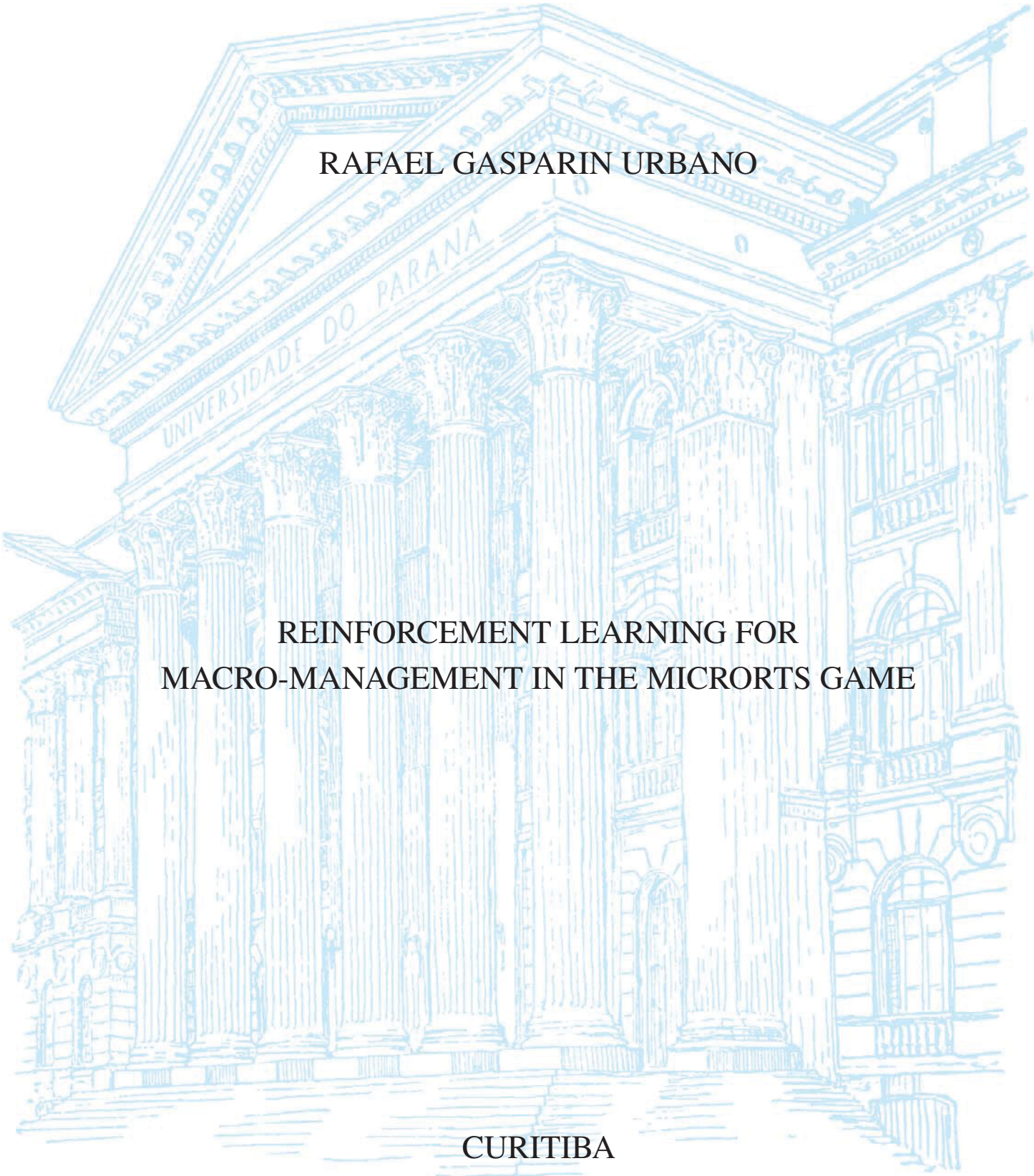
UNIVERSIDADE FEDERAL DO PARANÁ

RAFAEL GASPARIN URBANO



REINFORCEMENT LEARNING FOR MACRO-MANAGEMENT IN THE MICRORTS GAME

CURITIBA
2019

RAFAEL GASPARIN URBANO

REINFORCEMENT LEARNING FOR MACRO-MANAGEMENT IN THE MICRORTS GAME

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática, no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: David Menotti Gomes.

CURITIBA
2019

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

U72r Urbano, Rafael Gasparin
Reinforcement learning for macro-management in the microRTS game [recurso eletrônico] / Rafael Gasparin Urbano – Curitiba, 2019.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática.
Orientador: David Menotti Gomes.

1. Aprendizado do computador. 2. Inteligencia artificial. I. Universidade Federal do Paraná. II. Gomes, David Menotti. III. Título.

CDD: 004.8

Bibliotecária: Roseny Rivelini Morciani CRB-9/1585

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA - 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **RAFAEL GASPARIN URBANO** intitulada: **Reinforcement Learning for Macro-management in the MicroRTS Game**, sob orientação do Prof. Dr. DAVID MENOTTI GOMES, que após após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 26 de Setembro de 2019.

DAVID MENOTTI GOMES
Presidente da Banca Examinadora (UNIVERSIDADE FEDERAL DO PARANÁ)

JULIO CÉSAR NIEVOLA
Avaliador Externo (PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ)

MARCOS ALEXANDRE CASTILHO
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

FABIANO SILVA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

A minha mãe Susete e meu irmão Victor...

Agradecimentos

Agradeço a minha mãe Susete Gasparin Urbano, e meu irmão Victor Gasparin Urbano por todo apoio e amor durante esta jornada.

Aos meus avós Joana e Ivo, aos meus padrinhos Joy e Jamir, as minhas Tias Geovana e Elenice, aos meus tios Jovani e Marcos, aos meus primos, e em especial ao meu primo Douglas por todo carinho, compreensão e encorajamento para vencer esse desafio.

Aos amigos, em especial ao Andre, Anna, Ruanito, Rogério, Alessandro, Luiz e Adroaldo pro me ajudarem a trilhar esse caminho.

Ao meu orientador Prof. David Mennoti Gomes, por todo o suporte e incentivos nesse trabalho.

Ao Prof. Alexandre Ibrahim Direne, por despertar minha paixão por Inteligência Artificial já na primeira semana de aulas na UFPR.

Sem vocês não teria sido possível.

RESUMO

Nesta dissertação, apresentamos uma proposta para a aplicação de técnicas de aprendizado por reforço em jogos de estratégia em tempo real para aprender estratégias de macro-gerenciamento. Esta proposta tem como objetivo colocar técnicas comumente usadas em outros tipos de aplicações e mostrar como elas são mais rápidas e eficientes do que outras técnicas usadas neste campo. Para a implementação, usamos um jogo chamado MicroRTS, que tem as características mínimas para ser considerado um jogo de Estratégia em Tempo Real e foi desenvolvido para avaliar implementações de inteligência artificial sem precisar lidar com um jogo completo como StarCraft ou Wargus imediatamente.

Palavras-chave: Aprendizado de Máquina, Aprendizado por reforço, MicroRTS.

ABSTRACT

In this dissertation, we present a proposal for applying reinforcement learning techniques in Real Time Strategy games to learn macro-management strategies. This proposal aims to put techniques that are commonly used in other types of applications and show how it is faster and more efficient than other techniques used in this field. For the implementation, we use a game called MicroRTS, which has the minimal characteristics to be considered an RTS game and was developed to evaluate Artificial Intelligence implementations without having to deal with a full game like StarCraft or Wargus immediately.

Keywords: Machine Learning, Reinforcement Learning, MicroRTS, Macro-management.

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AI	Artificial Intelligence
RTS	Real Time Strategy
ML	Machine Learning
RL	Reinforcement Learning
DL	Deep Learning
GPU	Graphical Processor Unit
FPS	First Person Shooters
MOBA	Multiplayer Online Battle Arena
CNN	Convolutional Neural Network
API	Application Programming Interface
CBR	Case-Based Reasoning
ABCD	Alpha-Beta Considering Duration
MCTS	Monte Carlo Tree Search
MCTSCD	Monte Carlo Tree Search Considering Duration
DPF	Damage Per Frame
MLE	Maximum Likelihood Estimation
E-GAP	Extended Generalized Assignment Problem
BiCNet	Multiagent Bidirectionally-Coordinated Nets
SG	Stochastic Game
ALE	Arcade Learning Environment
DQN	Deep Q-network
ABL	A Behavior Language

GP Genetic Programming

AHTN Adversarial Hierarchical Task Network

HTN Hierarchical Task Network

Contents

1 INTRODUCTION

Thanks to advancements in technology, we are now able to do things that were just about impossible a few years ago. A wave of new gadgets and apps is flooding the consumer market, and Artificial Intelligence (AI) has now become the next big thing for them. Every day we read about a new self-driving car or a new personal assistant that promises to solve all of one's problems, even the ones that they did not even know they had. Companies are scrambling to find ways to put AI in all of their products. It's a revolution. Shortly, we will experience the changes that this will bring in our everyday lives.

In the front line of this revolution, we have Machine Learning (ML) with Deep Learning (DL) and Reinforcement Learning (RL), which consists of algorithms and techniques that can learn almost everything.

A DL algorithm consists of a set of training inputs that are processed through a relatively high number of layers, learning a deep network, with different functions to identify patterns. After the training, a network is generated with weights that can classify an input that was not in the training set.

A RL algorithm usually consists of choosing an action given a specific input. Using a reward function to calculate how good was the said action, so in the next time a similar input is presented it knows what action to make, and use this information to make improvements after each iteration.

1.1 Motivation

The scientific community has often used games to test their AI algorithms. Games like *tic-tac-toe* and *connect four* are probably a student first experience with AI. Board games have become an easy way to test new AI algorithms and techniques.

Researchers had some great success against professionals in some complex board games like *chess* [Campbell et al., 2002] and *go* [Silver et al., 2016]. For a while, the computational power was not enough to go through their large search space during a game. With the usage of Graphical Processor Units (GPUs) and their high processing power, suddenly, working with those large search spaces became viable. Algorithms proposed decades ago now are easier to run. Powerful GPUs opened many doors to researchers to explore domains that were unfeasible just ten years ago. After dominating the game of *go*, a great part of the community started to look for new challenges, and here is where we introduce computer video games.

Computer simulations have been used to train and demonstrate the capabilities of hundreds of different AI agents. Now, with video games that became even easier. We can now try to make AI agents win matches against humans. First, people started looking at simple games like pong, and it did not take long before AI agents were better than humans, some even achieving perfection. Then Mnih et al. [2015] dominated various different *atari* games. It has now became time to tackle really complex games like First Person Shooters (FPS) [Wang et al., 2009],

Multiplayer Online Battle Arena (MOBA) [Yang et al., 2014] and Real Time Strategy (RTS) games [Buro, 2004].

In recent events, a AI agent was capable of winning against human players in a computer MOBA called *Dota2* [Schulman et al., 2017], even though their bot had some limitations it was an impressive achievement. So far, only AplhaStar [Vinyals et al., 2019] has been able to beat professional players in more complex genre video games such as Real Time Strategy, like *StarCraft*. In the recent past big companies like *Google* [Vinyals et al., 2017], *facebook* [Synnaeve et al., 2016] and *Alibaba* [Peng et al., 2017a] all have started to research AI using RTS games.

Our motivation to use RTS games is due that they are very complex, Uriarte and Ontañón [2014] estimated in terms of search space that an game called *StarCraft* had a complexity that could range from $10^{50^{36000}}$ to $10^{200^{36000}}$ in a twenty-five minute game. In comparison, a game of chess has a complexity of 32^{80}, and a game of *go* ranges from 30^{150} to 300^{200}. Working in such a complex environment is complicated. Researching problems in this field can help to solve problems in different areas of study.

1.2 Challenges

There are several challenges in this field. There is uncertainty within the game when the Agent does not have all the information about the game state. Also, the real-time environment that has all the players executing actions at the same time. There is the problem of managing hundreds of units at the same time. Moreover, of course, there is a vast search space for those types of games. For this work, we are looking mainly at the search space and the *real-time* environment.

When creating strategies for a RTS game, we can divide them into two major groups, *Macro-management*, and *Micro-management*. Macro-management aims to control more generic and broad decisions like whether we should be attacking a position or defending it. Micro-management controls the details of the game, like the specific movement a unit must do to complete a task. The vast majority of work done in this field has explored the micro-management aspect of the game, especially the combat. The work that has touched on the macro-management uses a set of predefined strategies to choose. It remains a challenge learning micro-management and especially macro-management, with no previous knowledge about the game.

Another division we can make is on the type of learning. There is the prior learning where we need data collected from other agents or humans to train a new agent to play the game. For prior learning, the most considerable difficulty is to have an extensive data set for the training. Because of the high complexity of a RTS game, complete data set are not readily available. Then there is online learning, where an agent learns during the match. Here the problem is managing time. The longer an agent takes to make a decision, the further it is from actually playing in real-time. Furthermore, there is the inter-game learning, here the agent only learns between the games. Here the work has been sparse, but many of it has used some inter-game learning mixed with prior-game learning. Here the biggest challenge is creating an agent that can adapt to different situations.

These challenges and many others have fueled the research using video games. An algorithm that works in this environment has a great chance of working with other high complexity problems.

1.3 Proposal

We know from [Silver et al., 2017] that using RL techniques provides faster learning than using neural networks for board games. We try to demonstrate that the same holds in a more complex game, in this case, a RTS game called MicroRTS. Our focus is on making *macro-management decisions* in real-time using a combination of inter-game and online learning. We do not cover *micro-management decisions*, and we are assuming perfect information about the game state.

We propose to use RL techniques to create a learning agent capable of learning how to *macro-management decisions* and play a full game from zero knowledge. Our main objective is to show that our agent can learn how to make macro-management decisions and win against other AI implementations. To achieve that, we propose to use a RL to make actual decisions on the game. As our focus is not on the micro-management aspect of the game, we used programmed a set of hard-coded actions that our agent can do to perform the macro-management tasks.

This proposal aims to resolve two problems when developing AI to play RTS games. The first problem is the necessity of having an extensive data set to train our agent. The second problem is running the game in real-time.

Our evaluation uses two different parameters. The first is the effectiveness of this method when playing against other agents. These agents are different AI implementations provided by the scientific community, and it includes a range of agents, with some completely hard-coded, a few that use tree search algorithms, Monte Carlo algorithms, and more. First, we allow our implementation to play against each of the other agents so we can learn how to beat them individually after we play with them randomly to test whether we can learn to beat all of them using the same database. The second metric that we will use is the time it will take to train our agent.

1.4 Contributions

We demonstrated that our agent learns how to make *macro management decisions* with no prior knowledge of the game, using Reinforcement Learning techniques. The proposed Reinforcement Learning algorithm is based in Padmanabhan et al. [2015] with a few modifications (explained in Chapter 4). Also, as a secondary goal, we present lower training times and that our implementation uses fewer data to achieve the same if not better results.

1.5 Organization

This document is presented as follows, first in Chapter 2 we show some of the related work done with video games, focusing on RTS games. Later in Chapter 3 we explain what exactly is a RTS game, how we can learn it, and what we can expect from a AI agent in that environment as well as a look at the game we have used to run our experiments, the MicroRTS game. Then, in Chapter 4, we present our proposal in greater detail, including the reward function for our RL agent. Later in Chapter 5, we detail our experiments, and show and discuss our results. And finally, in Chapter 6, we draw our conclusions and suggest future works.

2 RELATED WORK

The research in this field is very young, and there are a lot of open challenges. In the beginning, researchers did not have access to the Application Programming Interfaces (APIs) to create agents, making it harder to test new implementations. To bypass this problem, clones of famous games were created with open APIs, like *Wargus*, which was based on the game called *Warcraft 2*. These clones, with their APIs, allowed more research to be done. Nowadays, we have official APIs of popular games like StarCraft 2, making the creation of new agents faster and more reliable. However, creating those agents still demands many hours of developing and training. Due to this difficulty, a researcher created the MicroRTS [Ontanón, 2013], a simple representation of a RTS game, aimed to make it easier to create and test agents before running it on more complex environments.

In the remainder of this chapter, we are going to explore some of the techniques that can be applied to the creation of the bots and the results that were already achieved by the scientific community.

2.1 Planning

One of the first works to explore this field was by Hsieh and Sun [2008]. In that work, the authors propose the usage of Case-Based Reasoning (CBR) to analyze human replays to learn how to play RTS games. They have focused their efforts on predicting *build orders*, i.e., in what order the agent will build or create units during the game. The CBR is used to resolve problems using knowledge from similar situations that the agent has already experienced. A ranking equation is used to obtain the degree of similarity between two states. The replays used are divided into two groups, one for training and the other for validating the training. The algorithms are composed of six steps, as shown in [Hsieh and Sun, 2008]:

1. Decomposing the inputted replay into states and strategies;
2. Using that to query the trained system;
3. Find equivalent states in the database and get its strategy. If there are no matches add to the database;
4. Calculate the score of each possible strategy, then choose the best;
5. Check if the choice is equal the current strategy, then go back to step 2;
6. Calculate the ratio of strategies accurately predicted.

Results showed that this system manages to learn how the player will behave, but due to limitations of not having a API, the learning was limited.

Weber et al. [2011] introduced a reactive planning agent (EISBot), for them achieving an excellent performance comes from being able to specialize in various competencies (*micro*) while working in high-level objectives (*macro*). Their objective was to build an agent capable of playing as well as humans, following the same restrictions. They explained that RTS games demand a non-heterogeneous architecture because different skills are needed for each aspect of the game. For them reducing the complexity of the domain is not trivial. More significant problems can be divided into subproblems, but there is an issue with this solution that is how to share the limited resources to each subproblem. The different idea was to create abstractions of the problems and use them to make decisions. Here the issue is creating a different abstraction to each one of the tasks needed to play the game. They proposed to divide the domain into individual competencies and create interfaces between them to resolve conflicts of the objective. The core agent is a reactive planner that supports real-time actions, and it is implemented in the A Behavior Language (ABL).

They divided the domain into five different competencies: Strategy, Income, Construction, Tactics, and Recon and called them managers. The integration between these managers and the core planner is done by augmenting the working memory, external goal formulation, external goal generation, and behavioral activation. This kind of integration can be used with other types of agents. What the agent does is the *micro-management*, the terrain analyses, the strategy selection, and deciding attack timing. Experiments have shown that the EISBot ranked better than 33% of humans, and achieved a winning record of 32% against them. When playing against bots, they had a 78% victory.

Ontanón and Buro [2015] proposed a technique that combines the minimax game tree search algorithm with the Hierarchical Task Network (HTN) planning, they called it Adversarial Hierarchical Task Network (AHTN). The algorithm assumes that there are two players. It searches a tree with a depth of d. Each node in this tree is a *touple* $\{s, N_+, N_-, t_+, t_-)$, where s is the state, N_+ and N_- are the HTNs plans for each player respectively, and t_+ and t_- represent execution pointers that keep track of which actions have been executed. They needed to extend the HTN to allow concurrent actions. They also defined five different definitions for the experiments using the MicroRTS game. They are, (1) Low level or AHTN-LL. It uses just primitive actions of that game. (2) Low Level with Pathfinding or AHTN-LLPF, changes the AHTN-LL by moving the units using the A* algorithm to find the shortest path from point a to point b. (3) A portfolio or AHTN-P has a set of hard-coded strategies to play the game, and it uses a portfolio search to select which one it will use. (4) Flexible or AHTN-F is the more elaborate method and uses only non-primitive tasks. (5) Flexible Single Target or AHTN-FST, it is similar to AHTN-F, but units attack the same target. The authors showed that their proposal was able to defeat various state-of-the-art implementations.

2.2 Tree search

Using a *tree search* algorithm, Uriarte and Ontañón [2014] focus on learning *micro-management* combat strategies in StarCraft. They used a matrix to represent the game state, where each line stands for each type of unit and region, and each column represents the player (which player controls this group), type (Type of units forming this group), size (number of units forming this group), region (which region is this group), order (which order is this group is currently performing), target (if the order requires, the ID of the target) and end (in which game frame is the order estimated to finish). For this abstraction, only one building is considered the base. The actions are N/A (for static units, e.g., base), Move (relocate to another region), Attack, and Idle (do nothing until the next search). Their model generates actions to be performed by

each group of units on the map. To choose the actions, they used a forward model that estimates how long the action will take, and what are the expected results.

The score is calculated as follows,

$$score = \sum_{1}^{n}(F_{i.size} \times F_{i.destroyScore}) - \sum_{1}^{m}(E_{j.size} \times E_{j.destroyScore}), \tag{2.1}$$

Where F is a friendly group of size n, E is a enemy group of size m and the destroyScore is the value of each unit. The bigger the score, the better is the action (Equation 2.1). Two different methods were implemented to evaluate this proposal. One using Alpha-Beta Considering Duration (ABCD), and the second a Monte Carlo Tree Search Considering Duration (MCTSCD) algorithm. They have disabled fog-of-war[1] and limited the maximum game time in 20 minutes. A new search is triggered every 400 logical frames. For the ABCD implementation, the depth of the search was limited in 3, the maximum number of children per node is 100.000, and the execution time limit for each search is 30 seconds. For the MCTSCD, the depth was limited to 10, 2000 payouts, and the game can last for up to 7200 frames.

For the experiments, the agent played 40 times in each map, against the built-in AI from the game. Results determined that the MCTSCD is slightly better than ABCD, but the authors concluded that the depth of the searches was not sufficient to achieve a good result in some cases. Aiming to fix these issues later that year, the authors released [Uriarte and Ontanón, 2014], with a proposal of three new abstractions to represent the game state. These new abstractions used more data to represent the state, mainly the information of all the buildings on the map and not just the base. The new tests showed that these new abstractions had better results than previous results.

Still trying to master the combat of the game, Uriarte and Ontanón [2015] proposed a way of learning the forward models automatically. They used graphs to model the map. Each region is a node, and the path connecting them are the edges. To simplify the unit representation, they were grouped up. Damage Per Frame (DPF) is how much damage a group of units can deal with within one frame of the game. The authors created two models to describe the combat in an RTS game: Sustained DPF: The damage is constant over time; Decreased DPF: The damage decreases over time.

Both models use the same three parameters. The unit hit point (how much damage it can sustain), unit DPF, and the target selection. The data set used to train was obtained from professional players replays. The actual combat was defined as a tuple $C = \langle t_s, t_f, R, U_0, U_1, A_s^0, A_s^1, A_f^0, A_f^1, K \rangle$, where t was the frame of the game when the combat started, R is the reason why it started, U_i is the upgrade list of the player, A_w^p where $w \in (s, f)$ is a representation of the army of the player, K is a list of frames where each unit died during the combat.

The main objective is to learn the DPF matrix. The results proved that this method works just as well as the models manually created to predict the combat, but it was faster. When paired with a Monte Carlo Tree Search (MCTS) algorithm, it works even better.

Another evolution to Uriarte and Ontañón [2014] can be found in [Uriarte and Ontañón, 2016], this time, the agent uses data from replays from humans experts to guide the search using Monte Carlo. The objective still focused on managing the combat of the game. The action *Move* proposed on the original work was expanded and now includes a type, it can be *ToFriend* or *ToEnemy* if the order is to move to a region that is occupied by our units or enemies units respectively, or *TowardsFriend* and *TowardsEnemy* if the Move is to a region where is adjacent to

[1]This is a mechanic of the game that prevents the player from seeing things on the map that are outside of the player's field of view

a region with friends or enemies. Those types are not mutually exclusive. A naive Bayes model is used to capture a probability distribution in which a human makes action given a specific game state. A Maximum Likelihood Estimation (MLE) is used to define the parameters of the model, which is used to set the default policy used by the MCTSCD. Results show a significant improvement over the previous version.

2.3 Multiagent

Using a *multiagent approach*, Tavares et al. [2014] proposes the use of *swarm* intelligence for the allocation of tasks in StarCraft. Their core agent aims to allocate multiple agents to do tasks they are fit to resolve, e.g., a soldier is more fit to attack an enemy than a worker. The implementation is based in Swarm-GAP, which in turn is an approximate algorithm to Extended Generalized Assignment Problem (E-GAP). The objective is that each agent will choose a task that needs to be done, then a global reward is calculated from the reward of each agent. This strategy needs a set of well-optimized parameters, for the StarCraft game, they defined 27 parameters. A genetic algorithm is used to define a combination of those parameters that maximize global performance. The input is an array with all the parameters.

A different approach to managing the combat in StarCraft is presented by Peng et al. [2017b], they also use multiple agents but in a coordinated network. The Multiagent Bidirectionally-Coordinated Nets (BiCNet) is formulated as a zero-sum Stochastic Game (SG). An SG of N agents and M opponents are described by the *touple* $\{S, \{A_i\}_{i=1}^{N}, \{B_j\}_{j=1}^{M}, T, \{R_i\}_{i=1}^{N+M}\}$ where S denote the state space. A_i is the action space of the agent i. B_j is the action space of the enemy j. $T : S \times A^n \times B^m \rightarrow S$ the deterministic transition function of the environment. $R_i : S \times A^n \times B^m \rightarrow \Re$ the reward function of the agent. The reward function is calculated for each agent, and the functions consider the other agents to force the network to learn how to cooperate. The results indicated that the BiCNet was able to learn complex strategies similar to the one adopted by human players. The fitness used is the final score of the game. The results are good, but not better than other implementations.

2.4 Adversarial Search

Going back to RTS games, Barriga et al. [2015] was not thrilled about the performance of the search algorithms proposed. So they introduced the Puppet Search. This approach is a framework based on scripts. The algorithm is a mechanism of abstraction of actions. Given a non-deterministic strategy, it chooses by picking actions based on the results of a look-ahead search. This strategy is given by a script that should be able to deal with every aspect of the game and expose choice points to the search algorithm. A script is a function that receives a game state and makes a decision, they can expose one or more choice points, called a puppet move, and each choice point generates a puppet search. The idea is to allow the look-ahead search to make crucial decisions based on its impact in the future, what the script does do not matter to the puppet search.

The implementation is based on a ABCD agent, with two modifications. One to consider a chain of puppet moves, and the other is that at any given point in time, two players can execute a puppet move. The algorithms disable fog-of-war. One puppet move may contain more than on action. For the tests, they used a script containing four different strategies, and it is expected that the puppet search will switch between them if something is not working. For the search algorithm, it was used three simulators to estimate the outcome of an action. For each puppet

move, the game is frozen for 6 seconds. The results showed better adaptability when compared with DQN.

2.5 Genetic Programming

Using Genetic Programming (GP), Garćıa-Sánchez et al. [2015] proposed to create complete strategies for the game StarCraft. The agent called StarCraftGP uses evolutionary algorithms to create scripts in c++ that should be able to play the game. For each individual, in the population, the script is compiled and executed to calculate its fitness. Two different metrics were used to calculate fitness. One used the end game score, a second more complex one separates the military from economic success and is calculated after three matches against four different opponents. The genetic operators used by them are one and two-point crossover, Subgraph Insertion Mutation, Subgraph Removal Mutation, subgraph Replacement Mutation, insertion Mutation, removal Mutation, replacement Mutation, single Parameter Alteration Mutation and alteration Mutation. The results are promising as StarCraftGP can beat hard-coded strategies written by experts.

2.6 Deep Learning

Justesen and Risi [2017] proposes to create an agent that can learn how to make *Macro management* decisions using DL. They use a data set containing 789,571 state-action pairs (2005 matches). The architecture is a multi-layered network, with fully connected layers. The input is a Game state having information about everything a player has done or observed. The output is the probability of building each construction given the input. There are two ways that the network chooses the action, either is a Greedy selection or a Probabilistic selection where the network does not make the best possible action but instead makes the most probable given its training. The results showed that the network could predict the build 54.6% of the time on the top-1. When tested playing the game, the agent won 68 out of 100 matches, using the probabilistic selection.

2.7 Reinforcement Learning

Using RL, more specific Q-learning, and State–action–reward–state–action (Sarsa) Sethy et al. [2015] presents an architecture that does not need a forward model. The game that he uses is called BattleCity, a basic RTS game. The agent makes random actions and observes the outcome, with that a reward is used to calculate the Q-values to the state-action pair. Two reward functions were defined. A Conditional Reward Function and a Generalized Reward Function. Each iteration follows these steps:

1. An action is chosen following a policy
2. The action is executed
3. A reward is given and saved
4. Update the Q-values in the Q-table following the learning algorithm

The results show a high winning rate playing against scripted AIs.

2.8 Hybrid

Despite the fact that the game analyzed in [Guo et al., 2014] is not a RTS game, in that work is was shown that it is possible to use MCTS to play computer games in real-time. The proposal is to build agents using planning to provide data to a deep learning architecture capable of playing in real-time. They used the Arcade Learning Environment (ALE) as an environment to run their tests. The authors claimed that RL is promising for choosing policies to play the game and that the DL is suitable for the perception of the environment, so combining them should produce a good result. As a base for the implementation, they used an agent built with Upper Confidence Bound 1 applied to trees (UCT), with no training and two parameters, the number of trajectories, and the maximum depth. Using UCT alone took days for each iteration. So the authors proposed three methods: 1) UCT to CNN via Regression; 2) UCT to CNN via Classification; 3) UTC to CNN via Classification-Interleaved. For the network, it was used the same presented in [Mnih et al., 2015] for their Deep Q-network (DQN). Their results showed a significant improvement over other implementations.

Recently Vinyals et al. [2019] presented the *AlphaStar*, a bot created using both DL and RL to play against professional *StarCraft 2* players. The bot's behavior is determined by a deep neural network that receives input data from the game and generates an action to be executed in the game. This network was trained using data from human players. This agent was able to defeat the most advanced *AI* built-in the game 95% of the time.

Then the authors created a pool of agents based on this initial training, and they were put to play against each other, using RL to improve their abilities to play the game. As the league progressed, more agents created from the current agents and were introduced to the league. The authors claimed that this allowed robust strategies to be developed. They also made it each agent has a different objective, such as defeat one specific other agent or using a set of units or even defeating all other agents.

The bot was evaluated using a Protoss vs Protoss matchup on a specific map with no other limitation. The adversary was two professional StarCraft players, playing five games against each one of them. The bot was able to beat both players 5-0.

2.9 Final remarks

We can see that apart from a few new studies, the research on this subject has always used handcrafted strategies and actions to guide the bots during the games. Although this method can yield excellent results, it will invariably be limited by how well those hardcoded strategies work. When those bots are put against more dynamic opponents or even human opponents, they fail to perform.

When Vinyals et al. [2019] introduced *AlphaStar*, it was a huge breakthrough, given that they managed to defeat human experts. However, their agent went through hundreds of years of gameplay simulated over several high-performance servers to be able to get to their results. This is not a setup easy to replicate, so there is still a lot to be explored in this area. In table 2.1, we can see that the majority of research has been done using some search algorithms. We believe that RL can bring better results with no dependency on pre-designed strategies or supervised learning.

Planning	Hsieh and Sun [2008], Weber et al. [2011], Ontanón and Buro [2015]
Tree Search	Uriarte and Ontañón [2014], Uriarte and Ontañón [2016], Guo et al. [2014]
Multiagent	Tavares et al. [2014], Peng et al. [2017b]
Look Ahead Search	Barriga et al. [2015]
Genetic Programming	Garćıa-Sánchez et al. [2015]
Deep Learning	Justesen and Risi [2017]
Reinforcement Learning	Sethy et al. [2015]

Table 2.1: Summary of works

3 BACKGROUND

In this chapter, we show a brief introduction to RTS games. After we explain what a RTS game is, what are its the defining characteristics and why they are an excellent tool to AI research, latter, we explain the bases we are using in this work.

3.1 Real time strategy games

The first RTS games date back to the late ’80s. However, it was only in 1994 when Blizard Entertainment released the game *Warcraft: Orcs and Humans* that the genre became popular. The game evolved with time, adding more and more complexity to its structure. As the years went by, more people started paying them, and the more they played better they became. Today there are massive competitions with professional players from all around the world playing for millions of Dollars.

3.1.1 Gameplay

A big part of the success of RTS games came from a core design guide for their development, which is:

"Easy to play, Hard to master."

That means that the games are easy to get in for new players, and challenging for the more experienced public. Let us explain what a RTS game is. In this type of game, the main objective usually consists in, gathering resources, constructing buildings, creating an army, and finally defeating one or more opponents. Those are what we call *macro-management objectives*. Resources can be described as a monetary system. All the buildings, all the units cost some amount of resources. A unit type called worker is used to gather more resources. Buildings are static structures that allow the players to create different types of units, or gather a more specific kind of resource and even unlock some upgrades and special abilities to the player’s units. Moreover, finally, we have the units. Those can be workers, as stated above, but they can also be troops, units designed for combat. They can have different types of strengths and weakness.

RTS games often feature something called *fog-of-war*. That is a mechanic of the game that prevents the player from seeing things on the map that are outside of the player’s field of view, reducing the amount of information available. In Figure 3.1 the dark part of the screen is in the *fog-of-war*, the player does not know what is happening there.

Figure 3.1: Fog-of-War

For one to play a RTS game, it is required first that he/she choose or create a tactic, which is a basic outline of how they want to play. Once that is taken care of, the player needs to break down this tactic in *macro-strategies*. These are decisions that do not take in consideration little details about the game, for example, a macro-strategy is choosing to attack a group of enemy's units, here we do not care how our units are going to engage. Taking care of those little details are the job of *micro-strategies*. With them, we want to control exactly how our units are going to engage the enemy. For instance, if the player has mix melee and ranged units, it is good if we keep the melee units in front of the ranged units so they can protect each other.

Being able to manage both macro and micro strategies during the game is what differentiates an average player from a good player.

3.2 MicroRTS

MicroRTS was developed by Ontanón [2013] and is a game designed to enable the studies of AI algorithms in a simple RTS game. It is developed in java. Only basic units are available, but the set can be expanded to include new units. In Figure 3.2 it shows a representation of a game state.

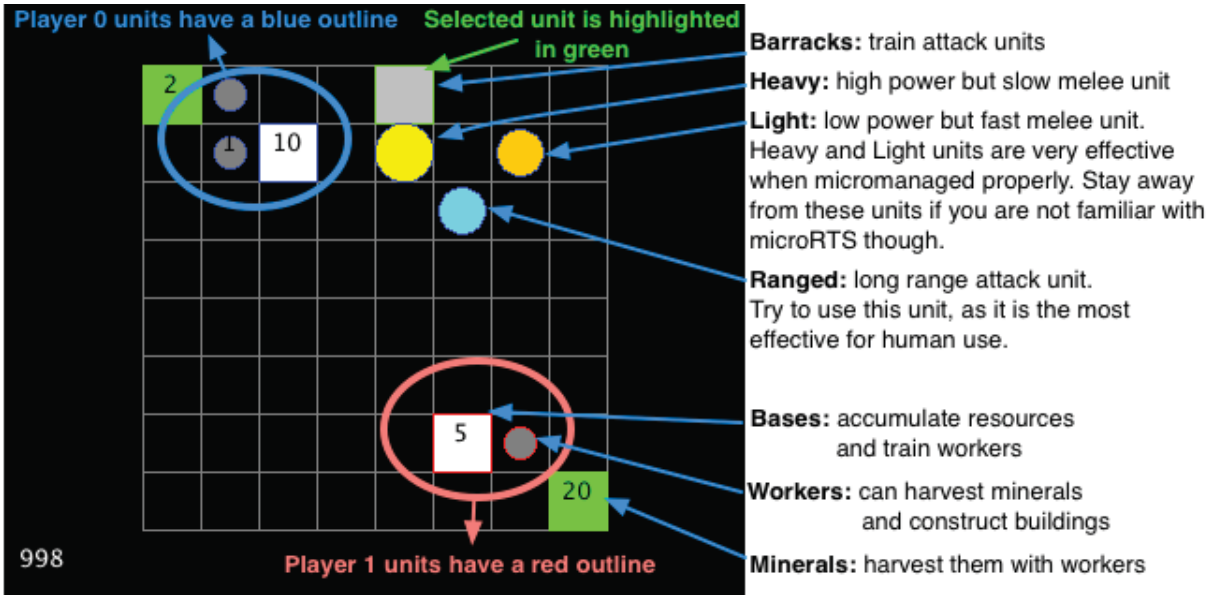


Figure 3.2: MicroRTS representation. Modified from [Stanescu et al., 2016]

3.2.1 Game state

The MicroRTS implementation represents the game state in four features. First is the time stamp that measures how many frames have passed since the match started. The second is called PhysicalGameState, and it has the map and every unit on it. The third is the Action assignments, which is a list of all the actions each unit is executing. And Finally is the Unit type table, that stores all the units that are configured to this match.

3.2.2 Actions

Each unit can only execute a subset of the possible actions. The actions are: Units can perform the following actions:

- None: the unit remains still for a certain number of cycles (specified as a parameter).
- Move: move one cell up, left, down, or right. Parameters:
- Attack_Location: attack a unit within range. The result is that the target unit loses hit points. Parameters: x,y coordinates of the attack location
- Harvest: collect resources form a resource source. Parameters: Direction: up, left, down, or right. (direction of where the resource source is)
- Return: deposit the resources the unit is carrying into a base. Parameters: Direction: up, left, down, or right. (direction of where the base is)
- Produce: create a new unit. Parameters: Direction: up, left, down, right. (the direction where the new unit will appear) Type: Base, Barracks, Worker, Light, Heavy. (the type of unit to produce)

MicroRTS also implements some abstractions of those actions, so the Agent does not need to deal with specific commands. The abstractions are:

- move: move a unit from coordinate x to coordinate y
- train: train a unit type
- build: build unit type in x, y coordinates
- Harvest: send a worker to harvest the resource and return to base
- attack: send a unit attack an enemy unit.
- idle: does nothing until the next frame.

With those abstractions the Agent does not need to issue basic actions to each unit, e.g., it does not need to move a worker to the coordinate to build a base, it can simply issue the worker o build the base at the coordinate.

3.2.3 Agents

In the default configuration, the game has two static units, a base used to store resources and create workers and a Barracks uses to train army units. The moving units are, workers used to gather resources and construct buildings, a light soldier with low cost but low hit points, a heavy soldier that is more expansive but has more hit points and ranged units that can attack from a distance. Each unit type has a set of moves that they can do.

The game can be configured to run as a deterministic game, fully-observable and real-time, but you can make it, so it is non-deterministic or partially observable.

The MircroRTS comes with several Agents implemented to help on the creation of new Agents. They are divided into several groups, we want to highlight some of them.

Using Hard-Coded Strategies, there are six different Agents implemented that are worth noting.

- RandomAI: Finds all possible actions for every unit, and chooses one action per unit to be executed in one frame.
- RandomBiasedAI: Follows the same logic as the RandomAI but has a bias to choose attack, harvest, and return.
- WorkerRush: Trains workers for the whole game, keep one harvesting and send others to attack.
- LightRush: Trains one worker to harvest, build one barrack and train light units for the rest of the game and send them to attack.
- HeavyRush: Trains one worker to harvest, build one barrack, and train heavy units for the rest of the game and send them to attack.
- RangedRush: Trains one worker to harvest, build one barrack and train ranged units for the rest of the game and send them to attack.

These Agents are fairly basic, but they can exploit some weakness on more complex Agents.

In the Portfolio Search category, there are two different versions. Both of them use a set of scripts that are used to control the Agent or each unit. The first version receives scripts with Hard-coded strategies that control the Agent and determine which one is better at run time

according to the game state. The second one gets scripts that can control each unit of the game. The Agent defines witch script each unit will use doing a greedy search.

Moving to Minimax Alpha-Beta Search strategies, MicroRTS has four Agents in this category. They use variants of the Real-time Minimax algorithm.

- RTMinimax: basic implementation of the Real-time Minimax algorithm. Depth is defined by time and not by a specific number of moves.
- IDRTMinimax: Similar to RTMinimax, but it uses all the available time to increase the depth of the search.
- IDRTMinimaxRandomized: Uses randomized alpha-beta search to better estimate the value of actions.
- ABCD/IDABCD: Modifies the tree to minimize the over or underestimation of the value of actions.

Using Monte Carlo Search Strategies, there are two basic strategies. The first considers every possible action and uses random simulations for each one. After all the simulations are done, a heuristic function is used to estimate the value of each action. The other uses Linear Side Information to reduce the search space.

For the UCT-based Strategies, there are three Agents. One is a basic UTC based on Real-time Minimax algorithm, other consider only a sample of all the possible actions at each node of the tree, and the last uses a tree uses only unit action when the other trees use Agent actions.

Applying Monte Carlo Tree Search Strategies, several Agents use different bandit strategies. There are two Agents in this group worth noting. One uses Match Learning with Polynomial Storage sampling strategy to choose the actions that will be considered. The other uses a naive-sampling idea from a naive Monte Carlo algorithm.

Using Hierarchical Task-Network Planning, there is only the Agent proposed in [Ontanón and Buro, 2015]. It uses minimax alpha-beta search with HTN planning.

4 PROPOSAL

In this chapter, we present our proposal for our Agent with the task of learning macro-management strategies for the game MicroRTS. The game we are using is a simple implementation of a RTS game. Figure 4.1 depicts how it works.

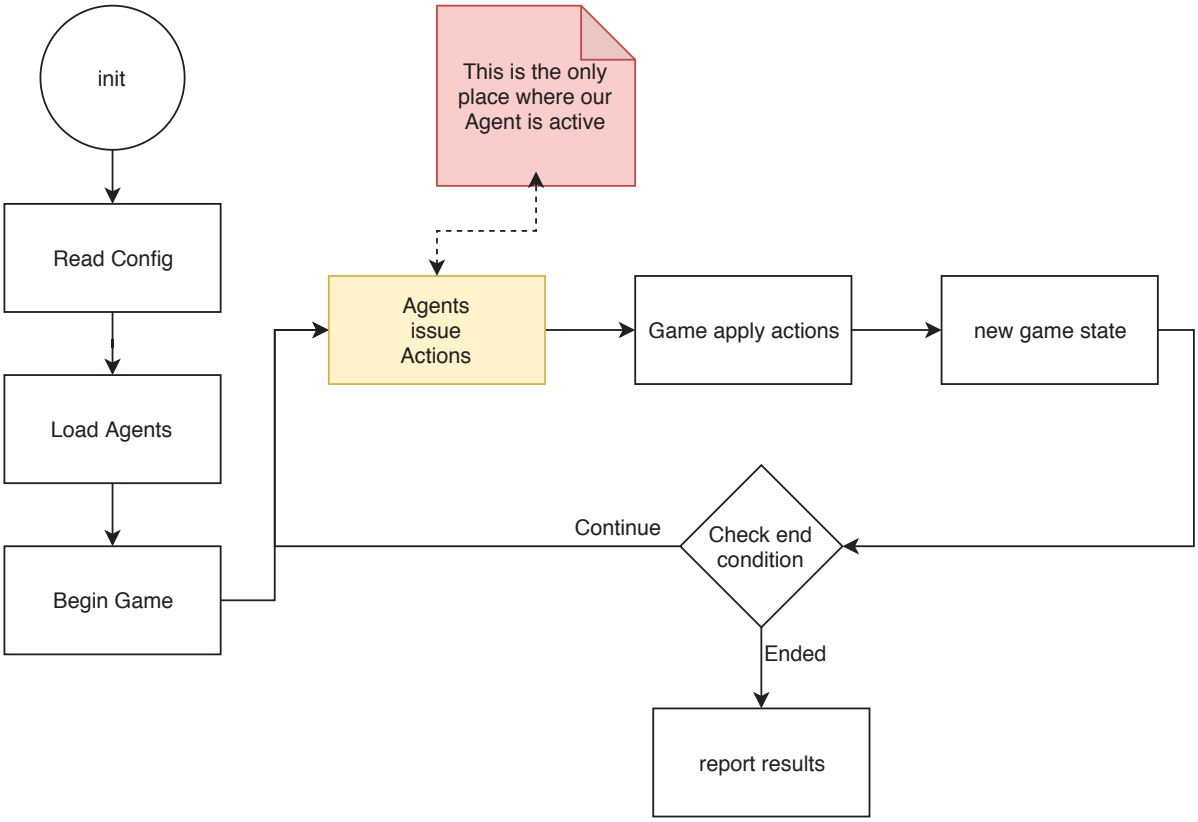


Figure 4.1: Basic flow of the game.

After the game loads its configuration and starts the match, the agents start to issue actions. They can issue at most one action per unit they have on the map, but several actions for multiple units can be taken in each frame. After the agents finish, the game executes all the actions, and then it generates the next frame, which is called the game state. The state is passed to the Agents to make new actions. This cycle repeats until an end game condition is detected, this condition can be a time limit or one Agent has defeated the other.

In the remainder of this chapter, we define the games state representation, the actions the agents can make, and finally, our Agent, with its RL algorithm and the similarity metric used.

4.1 Game state representation

The game state is a representation of everything that is happening on the game on a given frame. We represent our game state in a vector containing some of the data of the game. In this vector, it is stored how many units of each kind both players have, how many resources the Agent has, and how many of each building both players have. In total, the game state has fourteen different metrics. It contains the state of units, the number of resources and building, and also the actions that are under execution. Table 4.1 shows this representation.

0	Timestamp
1	FirendlyWorker
2	FriendlyLight
3	FirendlyHeavy
4	FriendlyRanged
5	FriendlyBase
6	FriendlyBarrack
7	EnemyWorker
8	EnemyLight
9	EnemyHeavy
10	EnemyRagned
11	EnemyBase
12	EnemyBarrack
13	Resouces

Table 4.1: Game state representation

Moreover, we also store the chosen actions on that frame, and its score. So that when this frame is chosen as the best one, its actions can be followed.

Using Figure 4.2 as an example, the game state would be represented by the Table 4.2. In the example, our Agent is represented by the color blue, and it is units have a blue outline. The greens squares are the resources that have not been collected yet, and the line on the units represent where they are looking. The white square is a base, and the gray square is a barrack, the small gray circle is a worker, and the medium teal circle is a ranged unit. Our Agent has one base, one barrack, and two workers, and our opponent has one base, one barrack, one worker and one Ranged unit and two resources stored.

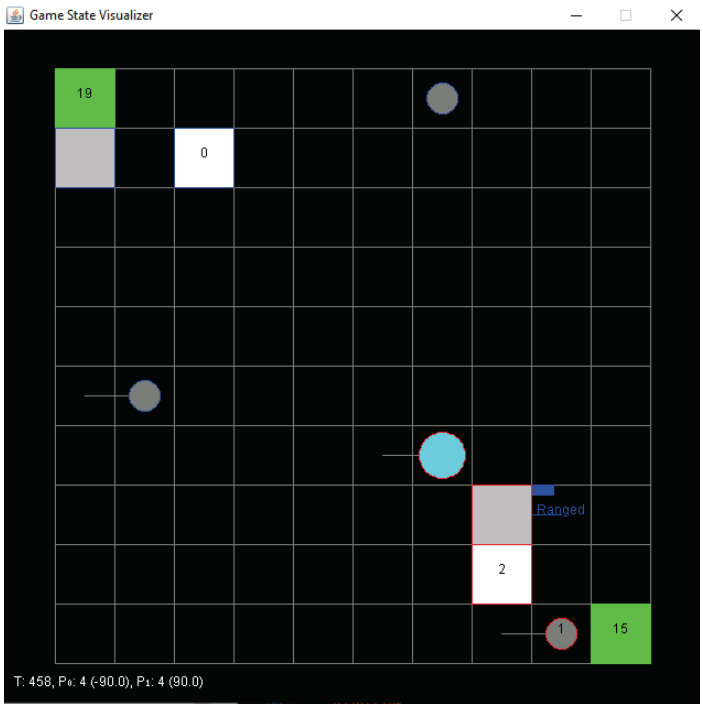


Figure 4.2: MicroRTS game.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Value	458	2	0	0	0	1	1	1	0	0	1	1	1	0

Table 4.2: Example of a game state representation

Note that we do not use spatial information on our Agent. Therefore we do not need to keep track of where each unit or building is located on the map. Although spatial information is crucial for the player, it would make it more troublesome for our Agent to compare states, so we have simplified our game state for that reason.

4.2 Actions (macro-management decision)

Our Agent can choose from a set of predefined macro-management actions. It is important to remember that a single macro-management action can trigger several micro-management actions. Those micro-management actions are hardcoded on our Agent.

The set of actions that our Agent can make is:

- BuildBase. Sends a worker to build a base;
- BuildBarracks Sends a worker to build a Barrack;
- TrainWorker. Trains a worker at the base;
- TrainLight Unit. Trains a Light attack unit at the Barrack;
- TrainHeavyUnit. Trains a Heavy attack unit at the Barrack;
- TrainRangedUnit. Trains a Ranged attack unit at the Barrack;

- Attack. Sends All units to attack the enemy base;
- Defend. Sends All units to defend the base;
- Gather. Sends a worker to gather resources.

The actions have some pre-requisites. The actions to train Light, Heavy, and Ranged units require that a Barrack be built beforehand. A base is necessary to train workers. The Attack and Defend Actions require that Barrack units have been trained. The build actions and gather action can only be performed by workers. To build a base or a Barrack or to train any unit, the Agent also needs the required amount of resources necessary to perform the action.

Some actions are mutually exclusive. Only one unit at a time for each Barrack or Base can be trained. All attack units must be performing the same action. Also, as stated above, a unit can perform only one action per time. So a worker can not Gather and build at the same time.

Many actions have a duration time. They are not instantaneous. The actions of building, training, and gathering have some duration. Therefore while the workers are building or gathering and the barracks or the base are training, they can not start another action.

4.3 Agent

Our Agent aims to learn how to create *Macro-management* decisions and develop strategies. Figure 4.3 shows a simple visualization of the proposed Agent.

It is learning by means of a RL algorithm that uses the game state as part of its reward function. We use inter-game learning to train our RL Agent and in-game learning to choose the best possible actions at any given time. *Inter-game learning* is used to calculate the scores after the game is done, and we know the results. Even though we are calculating the reward function, and the resulting score is stored during the game, the Agent will only have access to traces of the current game only in the next match. In fact, as implemented, in Inter-game learning, we only update the database with the traces of the last match played. *In-game learning* is used when the Agent is choosing which action it should make. In fact, here we use a greedy search strategy.

Although we are using a simple implementation of a Real Time Strategy game, we hope it can be carried over to more complex games like StarCraft.

In the interest to make our implementations more generic, it is out of our scope to learn the *micro-management* aspect of the game, which heavily depends on the game, and we are considering complete information by disabling the fog-of-war (defined in Section 3.1.1) in our experiments. As secondary objectives, we want to create a faster way of learning when there is no easy access to data to create a neural network, and the search space is too big to traditional AI.

Our main objective is to create an agent that is capable of learning how to play the game without having any prior training, i.e., the agent has no prior knowledge of how to play the game, but it knows the rules and all the actions it can issue.

The Agent starts by extracting the necessary information from the game state. It reads what units are alive on the map and how much resources it has accumulated. With that, it searches its database to look for similar values for those variables. If it fails to find any similar game state, the Agent issues a random action from all the possible actions. If it succeeds in finding one or more similar states, it executes the action(s) executed by the state with a higher score. After all the actions are executed, the Agent uses the new game state to calculate the reward function.

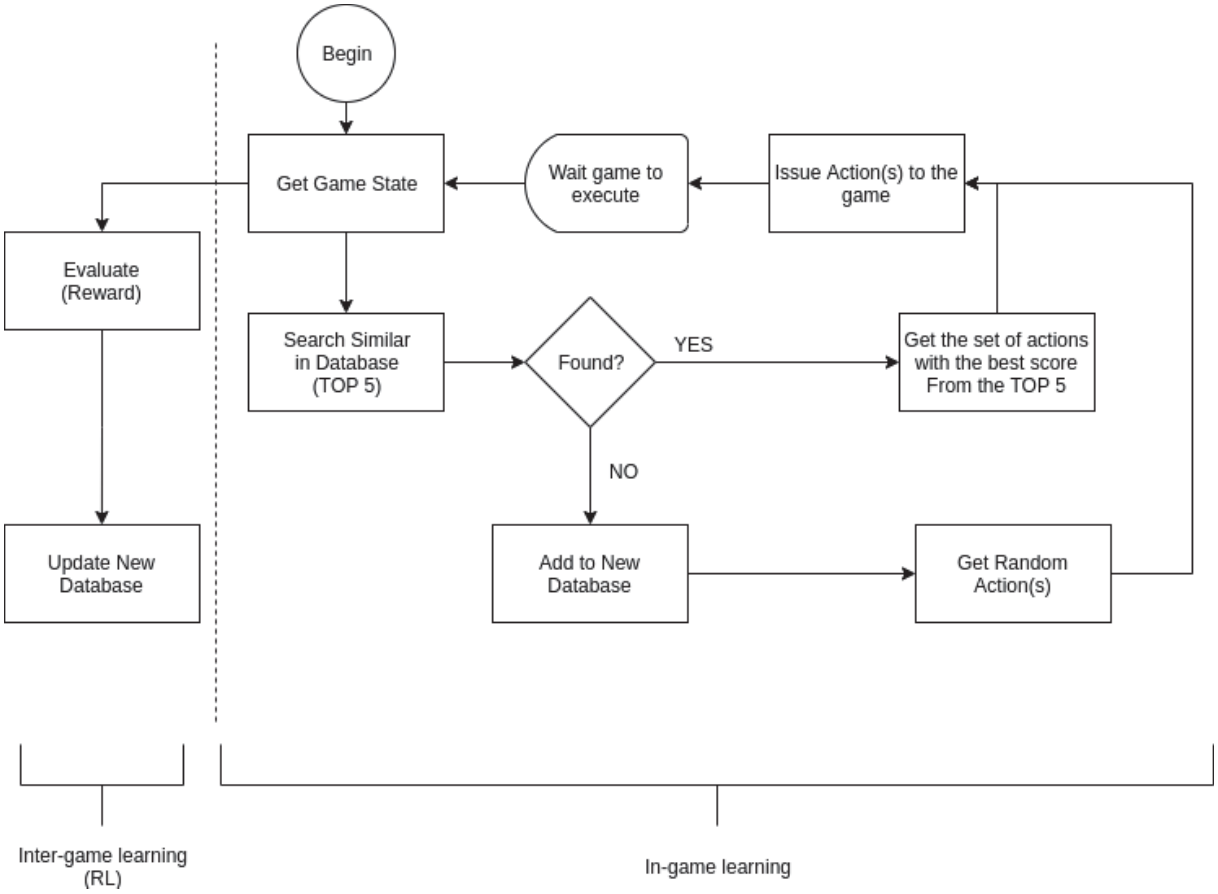


Figure 4.3: Basic flow of the agent.

4.3.1 Reinforcement learning algorithm

For the RL algorithm, we use a modified version of *IndividualActionPlanLearning* proposed in [Padmanabhan et al., 2015]. The original algorithm is not a RL, but we replace their forward model to introduce reinforcement learning. We choose this as our base because it showed a simple way to store and recover game states that can be used in real-time, and also as our objective is to play in real-time, we aim to use a simple method such as a greedy approach rather than using more time and resources for computing a better solution and not be able to do so in real-time.

The original algorithm uses traces extracted from previous matches to learn which action it should take in a specific state. It compares the state the Agent currently is with the states from the traces of its previous matches, it selects the top five most similar and puts these actions on a forward model to determine which one is the best for the current state. The more the Agent plays, the more knowledge it accumulates.

For our implementation we do not use a forward model (a simulation of the outcome of an action), we propose to use the experience our Agent accumulated from previous iterations to estimate this outcome in the form of a score, and after we have the actual outcome to calculate our reward. We propose to model this reward as:

$$Score = EconomicLoss_{enemy} - \alpha \times EconomicLoss_{agent} \tag{4.1}$$

where α value stands for the coefficient that weights the Agent's economic loss. The bigger the α figures are, the less aggressive are the strategies chosen by the Agent. $EconomicLoss$ is the

total value in resources that the Agent has lost. Economic loss is a metric that evaluates how cost-effective in combat the Agent has been. It is measured by calculating how much resource each unit that has been destroyed cost. Our Agent considers the whole match for this calculation.

The α defines how aggressive our Agent is. It does that by modifying the economic loss of our Agent, so it is more acceptable to lose units if we are killing some enemies in the process. Ideally, this parameter should be in-game adjusted to better react to different strategies. For instance when facing a hyper-aggressive Agent would be better if the α was a low value so our Agent would assume a more defensive stance at the beginning, but in this case, we want a higher value for α after the initial aggression has been dealt with to prevent our Agent from staying more passive.

This model is heavily focused on the results of combats by measuring how many resources each player lost, and we choose this due to our analysis, based on the literature review, that combat is the best way to measure success in this game. Nonetheless, if we focused only on resources, our Agent could be just gathering and not get close to the main objective of the game, which is to defeat the opponent. Moreover, if we had to use a model that used more variables, we would make our Agent more complex, and it might not have been able to play in real-time.

4.3.2 Similarity metric

Our Agent could calculate the similarity between two states by comparing the current game state with all the game states from the database. To reduce the number of states our Agent has to look through, and it only compares game states that were generated around the same time in-game. So, for example, if the current game state is frame 450, our Agent only looks for states that were generated 50 frames before and after that. We discuss this time window in Chapter 5.

Considering that our game state is a vector of size n, our similarity metric is the sum of the relationship R_i of each dimensional metric i-th between the current and any database game states, i.e.,

$$S = \sum_{i=1}^{n} R_i, \tag{4.2}$$

and R_i is defined as

$$R_i = 1 - \left| \frac{min(MC_i, MD_i)}{max(MC_i, MD_i)} \right| \tag{4.3}$$

where MC_i is the dimensional metric of the current game state and MD_i is the dimensional metric of any game state from the database, with $1 \leq i \leq n$ and $i \in \mathcal{N}$. Here, we defined $n = 14$.

Furthermore, each relationship R_i of our game states has to be smaller than a certain threshold. If the constraint is not kept for at least one dimension, the candidate state is discarded. This threshold defines how loosely the comparison between states is, and it is defined empirically in Chapter 5.

The final ranking of game state similarities to the current game state is given by sorting all game states inside the time windows such that we can choose only the top 5 most similar game states regarding the similarity metric S.

From the top five most similar game states, we choose the one with the highest score. Then, the actions performed by that state are recovered from the database, and the ones that can be executed are issued to the game. Note that the constraints to the execution of any action were described in Section 4.2.

5 EXPERIMENTS

In our experiments, we played our Agent against some of the implementations that were in the MicroRTS package. Unfortunately, none of the Agents used any learning, so we can not compare their training times with ours.

All the matches played with a maximum time limit of 5 000 frames. If this timeout is reached, it is considered a draw. We used a 10×10 map with no obstacles.

In the remainder of this chapter, we explain which Agents we used for our baselines, and we define the metrics we have used to measure our Agent, and also we define the fixed parameters we have used. Finally, we show the results of our experiments discussing their effects.

5.1 Implementation

We have implemented the agent using the Java programming language. We choose Java because the MicroRTS is also implemented in this language so it was easier to use their micro-magament implementation.

We implemented our Agent using three classes, MyBot, MyGameState, and RL. The main class is the MyBot, and it is responsible for the macro behavior of the units, generating and issuing all the actions to the game. To help translate the macro-management actions into micro-management actions, we have used an abstraction layer provided by the MicroRTS implementation.

The MyGameState class is the one that handles the game state. It has three main features, extracting our game state from the MicroRTS game state, calculate the similarity of the states and storing and updating them in the database.

Finally, the RL class handles the Reinforcement Learning algorithm. It uses the game state generated by the MyGameState class, calculates the reward function, and sends this information back to the MyBot class.

For our experiments, we have modified a test class provided by the MicroRTS code. It simply loads the map and the game configuration, then it creates an instance of each Agent and starts the game. It does this on a loop until some executions are done.

Being written in Java, our Agent is also multi-platform and can run on any machine with JRE 8 installed. We have also created scripts to help run experiments in a distributed manner. Multiple agents can collectively train using the same database.

Listing 5.1: MyBot

```
1 PlayerAction getAction(GameState)
2    currentGS <- GetMyGameState(GameState)
3    oldGS.score <- CalculateScore(currentGS, oldGS)
4    updateDatabase(oldGS)
5    if (!FindActionsFromTrace(currentGS ))
6       GenerateRandomActions();
```

```
7     SetBaseBehavior()
8     SetMeleeUnitBehavior()
9     SetWorkersBehavior()
10    SetBarracksBehavior()
11    return translateActions()
```

In Listing 5.1 we show a pseudo-code of our main class. The function *GetMyGameState* is implemented in the MyGameState class, and it translates the game state. The function *updateDatabase* is also implemented in that class and updates the score on the database. The *CalculateScore* is implemented in the RL class, and it calculates the reward function shown in 4.1. The *FindActionsFromTrace* is shown in Listing 5.2.

Listing 5.2: FindActionsFromTrace

```
1  boolean findActions(currentGS )
2     if(DatabaseIsEmpty())
3        return false
4     GameStateList <- Database.filter(TIME_WINDOW)
5     for each TraceGameState in GameStateList do
6        if (CalculateSimilarity(currentGS, TraceGameState))
7           add TraceGameState to GameStateCandidateList
8     TOP5GameStateCandidateList <- GetTOP5(GameStateCandidateList )
9     return getBestScore(TOP5GameStateCandidateList)
```

The *CalculateSimilarity* function uses the Equations 4.2 and 4.3

5.2 Enemies

In our experiments, the Agent plays against four hard-coded Agents introduced in Section 3.2.3 and the AHTNAI Agent introduced by Ontanón and Buro [2015]. These Agents are similar to each other because they all train only one worker to collect resources. However, they use different units to attack and are considered still challenging in different ways. Let us explain each one with their strengths and weaknesses.

- WorkerRush: Trains workers for the whole game, keep one harvesting, and send the others to attack. It is an extremely aggressive Agent and forces our Agent to deal with early aggression.
- LightRush: Trains one worker to harvest, build one barrack, and train light units for the rest of the game and send them to attack. This Agent is the one that uses the most cost-efficient units of the Hard-coded Agents. Light troops are reasonably cheap, and they overwhelm a worker defense.
- HeavyRush: Trains one worker to harvest, build one barrack, and train heavy units for the rest of the game and send them to attack. This Agent has the most expensive strategy requiring more resources to be collected, which requires more time, making the Agent's base more vulnerable to attacks during the unity training time, but heavy units are stronger than any other unit in the game.
- RangedRush: Trains one worker to harvest, build one barrack, and train ranged units for the rest of the game and send them to attack. This Agent is more efficient against passive Agents, forcing our Agent to be more aggressive.

- AHTNAI: Uses minimax alpha-beta search with HTN planning proposed by Ontanón and Buro [2015]. This Agent is the most complex of the ones we are using.

It is essential to observe that although we are evaluating our Agent against only five hard-coded Agents, our Agent does not know its enemy.

5.3 Metric evaluation

We use two different metrics to evaluate our implementation. First and foremost, we want to know the effectiveness of our Agent, so we compute the win ratio of the last 20 games, and we report only the initial and final win ratio, that is, the win ratio of the first and last 20 games, respectively.

The second metric is how much time each match takes. Although we do not have any other learning Agent to compare with, we aimed to make the time of each match takes consistent with the size of the database.

5.4 Parameter definitions

As explained in Chapter 4, we have a few parameters in our algorithms that we have empirically defined. In this section, we explain how we have defined those parameters.

The first parameter that we have discovered was the time window in which we look for similar game states. This parameter is critical to help reduce the search space. It also helps to develop strategies that take into consideration the passage of time during the game. In games, time is usually measured in how many frames have been processed since the beginning of the match.

Real Time Strategy Games have a clear timeline during a match, and it can be divided into the early game, middle game, and finally, the late game. Each of those time frames has different strategies. We have decided to keep our time window relatively small, with a size of 30 frames. RTS games usually are played by humans at 60 frames per second, and the average professional RTS pro player executes four actions per second. Therefore a time window of 30 frames allows or Agent to for game states that are one or two actions of difference. Moreover, if this window were higher than 30 frames, the Agent would have less time to respond to enemy actions, and if it were lower, the Agent would not have time to complete actions or to make a significant change to the game state and would choose the same actions again.

The second parameter of our algorithm is the α value in Equation 4.1. As previously explained, this parameter is a modifier to the value of the Agent's economic loss. It allows us to tweak the aggressiveness of the Agent. After some experimentation, we have reached a value of 0.85, and it allowed the Agent to fend off some of the aggressive enemies but not be completely passive. The reason that we do not change this value during the game is to maintain a consistent score for our In-game learning mechanism to accurately compare different scores from different game-states. If we desired to change this value during training, we would need a more complex structure to allow the comparison of the different scores generated with different α, but this is not in our scope.

Lastly, there is a similarity threshold. This value defines how similar two metrics need for the Agent to consider it as a candidate. This value was initially defined by [Padmanabhan et al., 2015] as 0.21. The authors stated that if this value were > 0.21 to find the top 5 game states would take to long, and if it were < 0.21, the states would no be similar enough. We have found that this value is also beneficial for our algorithm.

5.5 Results and discussions

Our results prove that this proposal has the potential to be used to create basic strategies, but it fails to develop more sophisticated strategies that would allow it to defeat more complex opponents.

Our Agent trained against each of the enemies individually.

Table 5.1 shows the number of games our Agent played against each Agent during the training process, how many different game states were generated, as well as the win rate of the first and last 20 games.

Agent	#Games	#GameStates	Begin Win Rate	End Win Rate	Max Win Rate
WorkerRush	19927	3473	0%	0%	0%
LightRush	7171	29597	75%	95%	100%
HeavyRush	4496	38760	5%	75%	100%
RangedRush	13130	14474	70%	100%	100%
AHTNAI	8208	11785	0%	0%	0%

Table 5.1: Summary of the experiments

Our Agent was not able to win any games against the WorkerRush nor the AHTNAI Agents. When we analyzed the data, we have found that both of those Agents played very similarly, using one worker to collect resources and using the rest of the units to rush our Agent's base. The AHTNAI Agent has more complex strategies that it can use, but it always starts with the same strategy as the WorkerRush Agent. Our proposal could not develop any strategy to be able to deal with this kind of early aggression. We also tried changing some of the parameters to see if it would have an impact on the training of our Agent. We changed the α to 1.2 to create a more passive behavior. We also changed the time window to 3 frames, in hopes it would help the reaction time against these kinds of hyper-aggressive Agents. Unfortnly it had the same result, after 2000 games of training our Agent was still unable to win a single one of them.

Moving to the other Agents, let us analyze them one by one. We are starting with the HeavyRush Agent. Our Agent won 75% of the last 20 games of the training. However, it is interesting to note that during the training, our Agent managed to have a 100% win ratio on 20 consecutive games a few times during the experiment. The first time our Agent managed to have that was after only 202 games were played. However, after game 255, our ratio was at just 60%. This fluctuation continued to happen during the whole training. Figure 5.1 shows the moving average of the last 20 games at any given time.

We can see that our Agent was able to learn and re-learn how to win against this Agent several times. The reason that it was not possible to find a stable strategy is that the way we have used the micro-management was random at times, and against an Agent that uses the most powerful unit of the game to attack, any mistake can be lethal. So, we were choosing the best possible action, but our execution of that action was far from perfect, causing a low score. Therefore our Agent considered an Action that once was good, a bad one, and abandoning it, essentially resetting the progress and forcing it to find a new way to defeat the opponent.

Now let us analyze the training against the RangedRush Agent.

We can see from the graph shown in Figure 5.2 that we have a similar situation of what happened with the HeavyRush training. For a few times, our Agent reached a 100% win rate on the moving average of the last 20 games, and again, our Agent had to re-learn how to beat the enemy. However, this time after the 725th game, our Agent managed to find a stable strategy and consistently defeat the RangedRush Agent. The difference is that the unit the opponent was

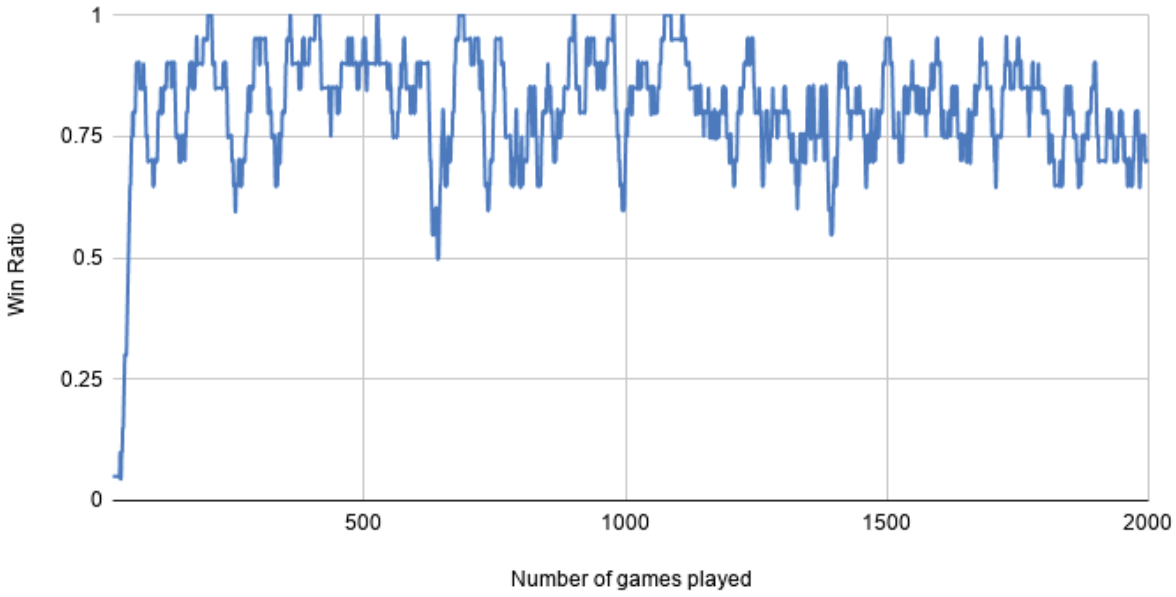


Figure 5.1: Moving average of the last 20 games against the HeavyRush Agent.

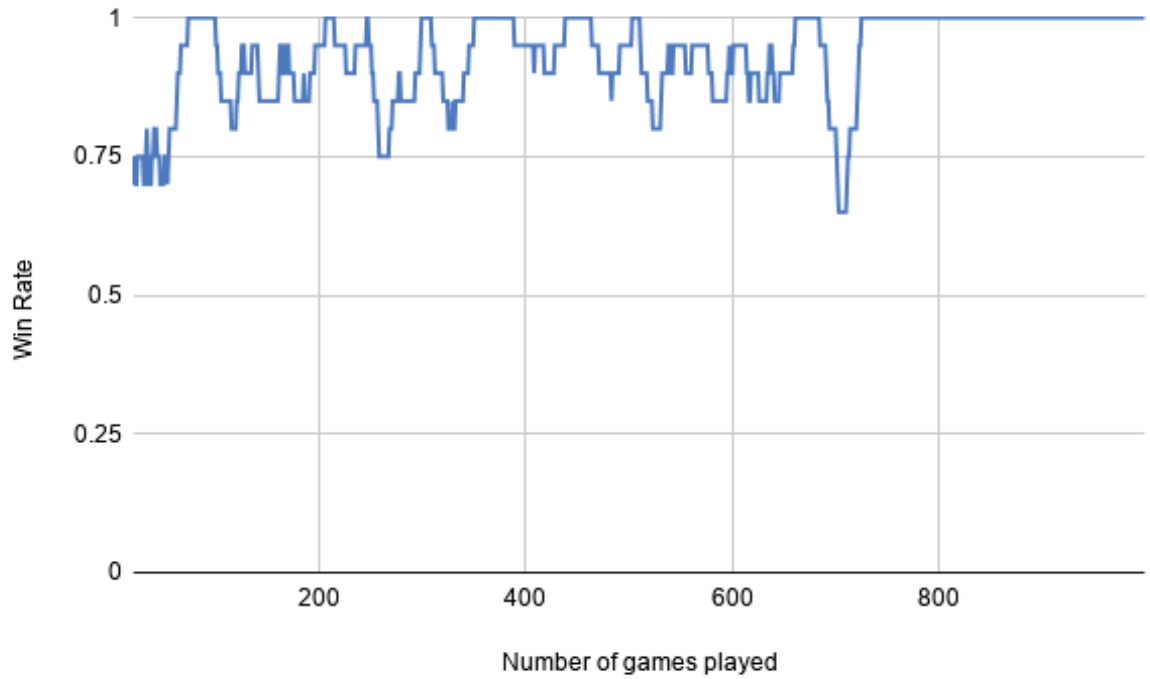


Figure 5.2: Moving average of the last 20 games against the RangedRush Agent.

using, the ranged unit, deals less damage and is easier to kill than the heavy unit that the other Agent was using, making our little imperfections on the execution of the actions less disastrous.

Against the LightRush enemy, we observed the same pattern again. Figure 5.3 shows a graph that illustrates how the training progressed.

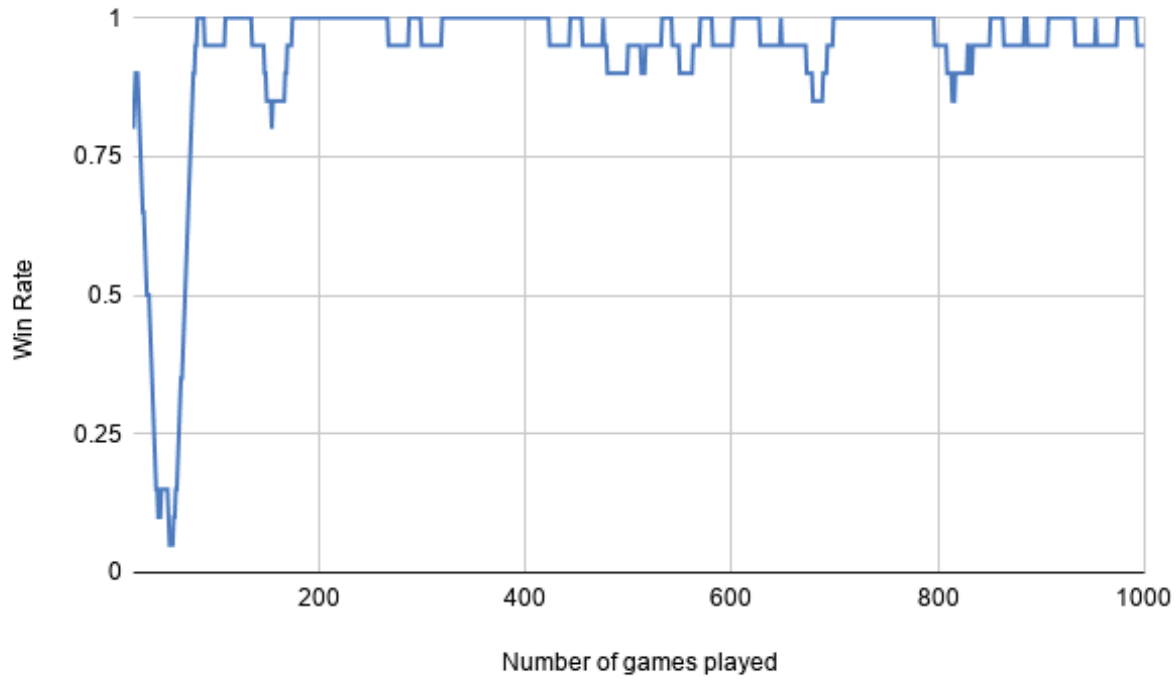


Figure 5.3: Moving average of the last 20 games against the LightRush Agent.

After our Agent played 89 games, it achieved the 100% win rate. However, it was not able to sustain that win rate. The reason is the same as for the training against the HeavyRush Agent, their unit is cheap, and they can quickly punish the mistakes of our Agent.

We also tried to use the database used to train our Agent against one of the enemies and use it to play against another one. As we have five trained databases and five Agents, we had a total of 20 combinations and played 20 games for each one of them. We have found that the game states generated were too different to enable our Agent to use any of the experience it accumulated playing a different Agent. It behaved like it was the beginning of the training against that specific Agent.

Finally, we have made an analysis on the time each game took, before and after the training. The results are shown in table 5.2.

Agent	Inital Time AVG	Final Time AVG
WorkerRush	$7,25s$	$5,15s$
LightRush	$12,1s$	$20,9$
HeavyRush	$21,6s$	$39,75s$
RangedRush	$13,85s$	$6,65$
AHTNAI	$7,05s$	$9,7s$

Table 5.2: Time analysis of match duration against each Agent

We can see that for the training against the RangedRush Agent, our Agent is able to have a better time than it had at the begging of the training. That happened due to the fact the for that Agent, our proposal was able to find an optimal strategy. However, for the Agents LightRush and HeavyRush our Agent is still learning so it takes more time for it to find the actions it should make.

For the WorkerRush Agent, the times became faster due to the failed maximum local strategy that keeps our Agent from winning any matches. Finally, the times Against the AHTNAI have increased because their implementation uses different strategies, so it can take more, or it can take less time to defeat our Agent.

6 CONCLUSION

In this work, we have proposed an Architecture to enable an Agent to play the MicroRTS game, creating Macro-management strategies using an Reinforcement Learning algorithm. Our Agent had no previous knowledge of how to create those strategies. During our experiments, we have found that although our implementation is, in fact, able to learn, it could not defeat some hyper-aggressive and more complex Agents.

Several improvements could be made to make our implementation more efficient. To start, a better micro-management implementation than the one that we borrowed from the MicroRTS implementation. Analyzing the data, we have noticed that some times our workers would obstruct the path to the resources with a base or a barrack, preventing them from continuing to collect them. Another issue that we have found is that our units were not focusing on attacking the same enemy unit. The way we have coded our attacking behavior was to go for the nearest enemy instead of sending all units to attack the same target.

Secondly, using spatial information on our game state. We have also noticed our units were spread on the map and with no cohesive group formation, our units were vulnerable to be isolated and destroyed. With spatial information, we could use the same strategy to our advantage an isolate and destroy the enemy units.

Also, our greedy approach for finding the best possible action was getting stuck in local maximum values, as expected. Using some strategy to find the best actions that allow for some not so good ones might lead to a better overall strategy.

Finally, our implementation does not take into consideration the actions made previously in the game, so only the current state matters. Creating a sequence of steps that are applied to multiple states could also help to develop more sophisticated strategies.

6.1 Future work

For the future, we want to explore more variables and parameters of our Agent. Learn how different values for our time windows, for the α value, and similarity threshold affect the training and results of our Agent.

We also want to work on micro-management actions. We believe that mastering the little details of the game would improve the capacity of our Agent of creating macro-management decisions. On There is also more research to be done with deep neural networks, in both macro and micro-management tasks. As neural networks are great to detect patterns, it could help predict actions of the enemy, giving valuable information to the Agent.

REFERENCES

Nicolas A Barriga, Marius Stanescu, and Michael Buro. Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games. In *Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 9–15, 2015.

Michael Buro. Call for ai research in rts games. In *Proceedings of the AAAI-04 Workshop on Challenges in Game AI*, pages 139–142. AAAI press, 2004.

Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

Pablo Garćıa-Sánchez, Alberto Tonda, Antonio M Mora, Giovanni Squillero, and JJ Merelo. Towards automatic starcraft strategy generation using genetic programming. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 284–291. IEEE, 2015.

Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in neural information processing systems*, pages 3338–3346, 2014.

Ji-Lung Hsieh and Chuen-Tsai Sun. Building a player strategy model by analyzing replays of real-time strategy games. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 3106–3111. IEEE, 2008.

Niels Justesen and Sebastian Risi. Learning macromanagement in starcraft from replays using deep learning. *arXiv preprint arXiv:1707.03743*, 2017.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Santiago Ontanón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

Santiago Ontanón and Michael Buro. Adversarial hierarchical-task network planning for complex real-time games. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

Vineet Padmanabhan, Pranay Goud, Arun K Pujari, and Harshit Sethy. Learning in real-time strategy games. In *Information Technology (ICIT), 2015 International Conference on*, pages 165–170. IEEE, 2015.

Peng Peng, Quan Yuan, Ying Wen, Yaodong Yang, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent bidirectionally-coordinated nets for learning to play starcraft combat games. *arXiv preprint arXiv:1703.10069*, 2017a.

Peng Peng, Quan Yuan, Ying Wen, Yaodong Yang, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent bidirectionally-coordinated nets for learning to play starcraft combat games. *arXiv preprint arXiv:1703.10069*, 2017b.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL `http://arxiv.org/abs/1707.06347`.

Harshit Sethy, Amit Patel, and Vineet Padmanabhan. Real time strategy games: a reinforcement learning approach. *Procedia Computer Science*, 54:257–264, 2015.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

Marius Stanescu, Nicolas A Barriga, Andy Hess, and Michael Buro. Evaluating real-time strategy game states using convolutional neural networks. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–7. IEEE, 2016.

Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. Torchcraft: a library for machine learning research on real-time strategy games. *CoRR*, abs/1611.00625, 2016. URL `http://arxiv.org/abs/1611.00625`.

Anderson R Tavares, Hector Azpúrua, and Luiz Chaimowicz. Evolving swarm intelligence for task allocation in a real time strategy game. In *Computer Games and Digital Entertainment (SBGAMES), 2014 Brazilian Symposium on*, pages 99–108. IEEE, 2014.

Alberto Uriarte and Santiago Ontañón. Game-tree search over high-level game states in rts games. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

Alberto Uriarte and Santiago Ontanón. High-level representations for game-tree search in rts games. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

Alberto Uriarte and Santiago Ontanón. Automatic learning of combat models for rts games. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

Alberto Uriarte and Santiago Ontañón. Improving monte carlo tree search policies in starcraft via probabilistic models learned from replay data. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M. Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Yuhuai Wu, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

Di Wang, Budhitama Subagdja, Ah-Hwee Tan, and Gee-Wah Ng. Creating human-like autonomous players in real-time first person shooter computer games. In *Proceedings, Twenty-First Annual Conference on Innovative Applications of Artificial Intelligence*, pages 173–178, 2009.

Ben George Weber, Michael Mateas, and Arnav Jhala. Building human-level ai for real-time strategy games. In *AAAI Fall Symposium: Advances in Cognitive Systems*, volume 11, page 01, 2011.

Pu Yang, Brent E Harrison, and David L Roberts. Identifying patterns in combat that are predictive of success in moba games. In *FDG*, 2014.